# CPSC 330
# Applied Machine Learning

## Lecture 18: Multi-class classification and introduction to computer vision

UBC 2022-23

Instructor: Mathias Lécuyer

## Imports

In [2]:
```
%load_ext autoreload
%autoreload 2
```

In [3]:
```
import glob
import os
import sys
sys.path.append("../code/.")

from mglearn_utils import discrete_scatter, cm3, plot_2d_classification
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn import datasets
from sklearn.dummy import DummyClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    plot_confusion_matrix,
)
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.svm import SVC
```

## Learning objectives

- Apply classifiers to multi-class classification algorithms.
- Explain the role of neural networks in machine learning, and the pros/cons of using them.
- Explain why the methods we've learned previously would not be effective on image data.
- Apply pre-trained neural networks to classification and regression problems.

- Utilize pre-trained networks as feature extractors and combine them with models we've learned previously.

## Multi-class, meta-strategies

- So far we have been talking about binary classification
- Can we use these classifiers when there are more than two classes?
  - "ImageNet" computer vision competition (http://www.image-net.org/challenges/LSVRC/), for example, has 1000 classes
- Can we use decision trees or KNNs for multi-class classification?
- What about logistic regression and Linear SVMs?

- Many linear classification models don't extend naturally to the multiclass case.
- A common technique is to reduce multiclass classication into several instances of binary classification problems.
- Two kind of "hacky" ways to reduce multi-class classification into binary classification:
  - the one-vs.-rest approach
  - the one-vs.-one approach

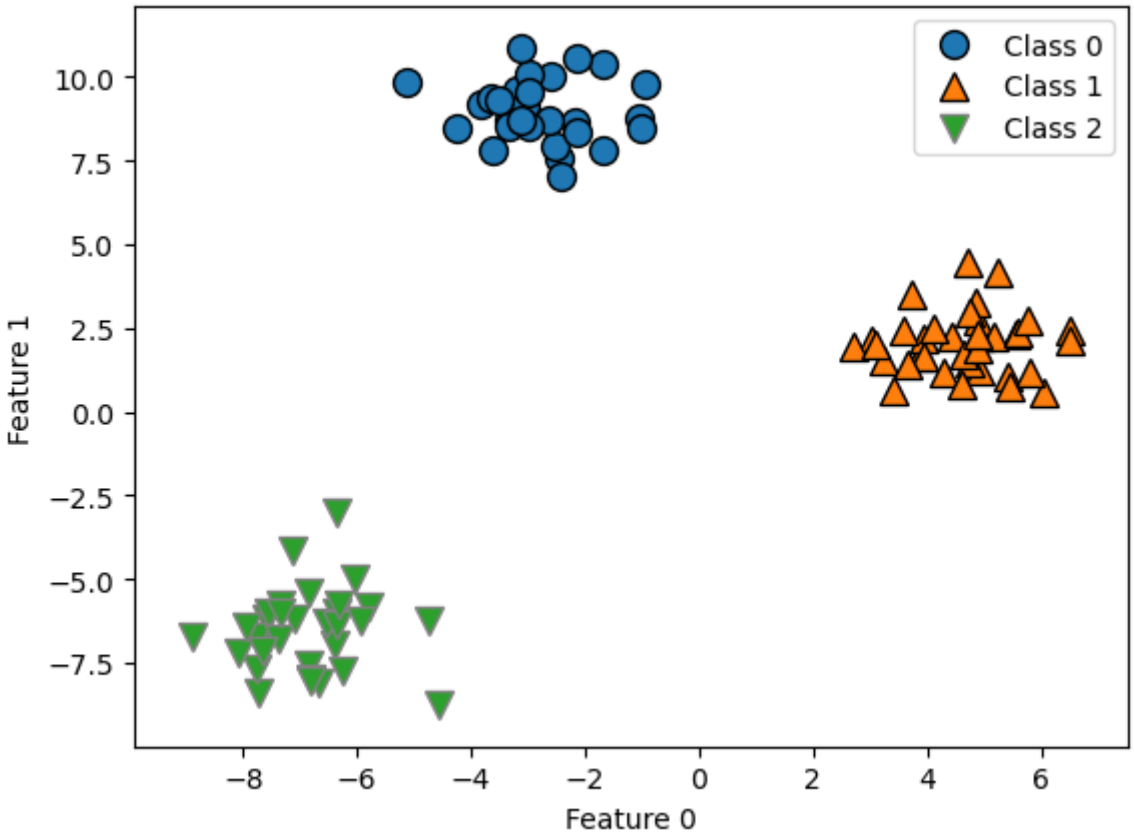
### One vs. Rest

- 1v{2,3}, 2v{1,3}, 3v{1,2}
- Learn a binary model for each class which tries to separate that class from all of the other classes.
- If you have $k$ classes, it'll train $k$ binary classifiers, one for each class.
- Trained on imbalanced datasets containing all examples.
- Given a test point, get scores from all binary classifiers (e.g., raw scores for logistic regression).

- The classifier which has the highest score for this class "wins" and that's going to be the prediction for this class.
- Since we have one binary classifier per class, for each class, we have coefficients per feature and an intercept.

Let's create some synthetic data with two features and three classes.

In [4]:
```
from sklearn.datasets import make_blobs

X, y = make_blobs(centers=3, n_samples=120, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=123
)
discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(["Class 0", "Class 1", "Class 2"]);
```




In [5]:
```
lr = LogisticRegression(max_iter=2000, multi_class="ovr")
lr.fit(X_train, y_train)
print("Coefficient shape: ", lr.coef_.shape)
print("Intercept shape: ", lr.intercept_.shape)
lr.coef_
```

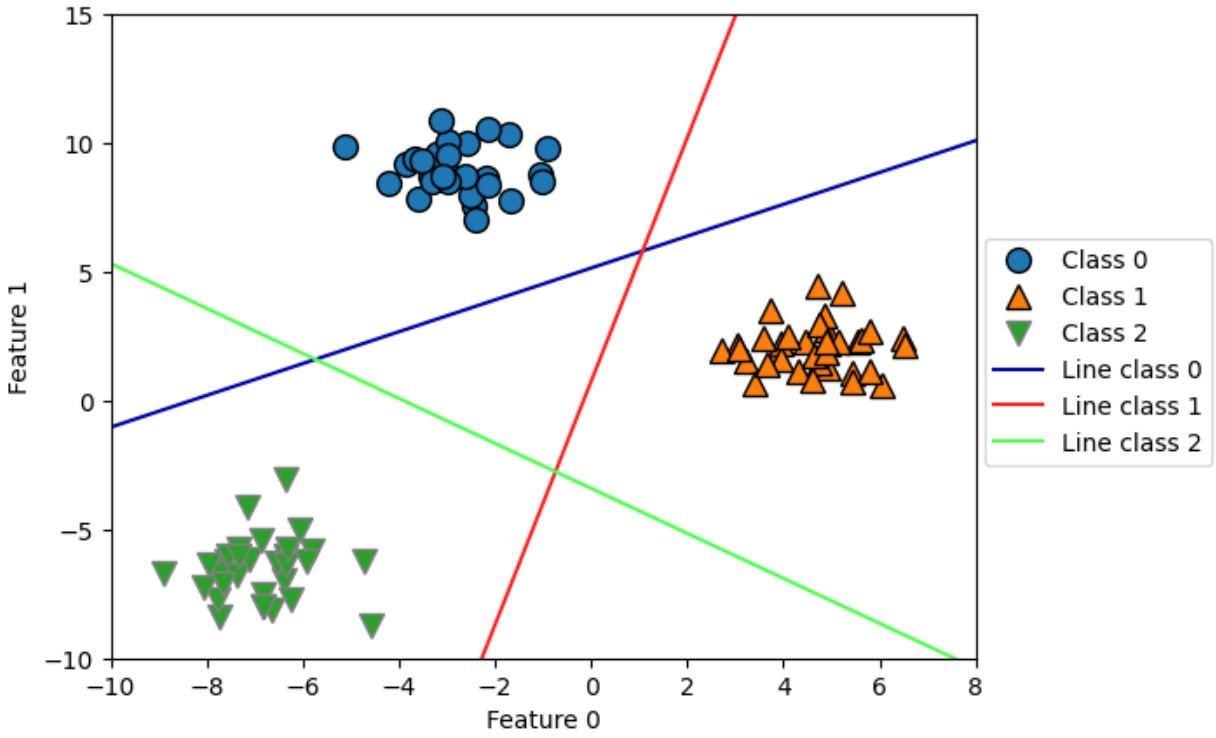
```
Coefficient shape:  (3, 2)
Intercept shape:  (3,)
```

Out[5]:
```
array([[-0.65123329,  1.0536117 ],
       [ 1.35418019, -0.28647501],
       [-0.63320669, -0.72513556]])
```

- This learns three binary linear models.
- So we have coefficients for two features for each of these three linear models.
- Also we have three intercepts, one for each class.

Code credit (https://learning.oreilly.com/library/view/introduction-to-machine/9781449369880/ch02.html#linear-models)

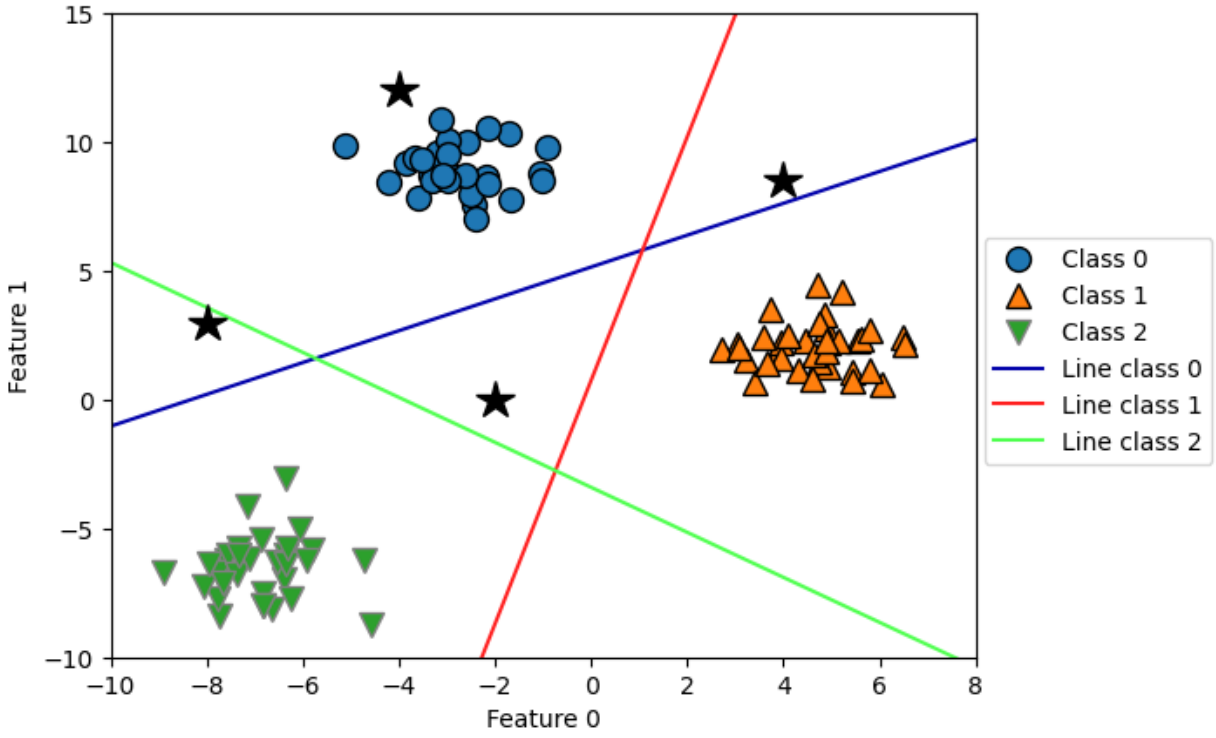
In [6]:
```
discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(lr.coef_, lr.intercept_, cm3.colors):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.ylim(-10, 15)
plt.xlim(-10, 8)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(
    ["Class 0", "Class 1", "Class 2", "Line class 0", "Line class 1", "
    loc=(1.01, 0.3),
);
```



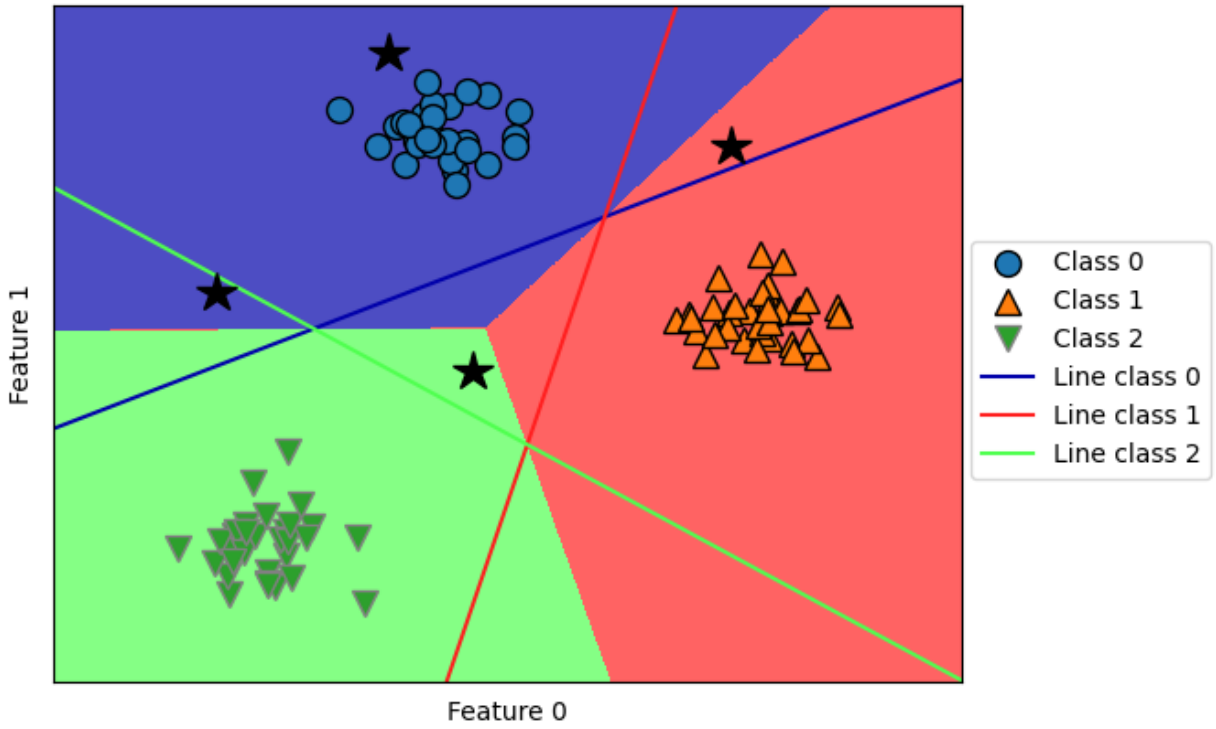

In [7]:
```
def plot_test_points():
    test_points = [[-4.0, 12], [-2, 0.0], [-8, 3.0], [4, 8.5]]
    plt.plot(test_points[0][0], test_points[0][1], "k*", markersize=16)
    plt.plot(test_points[1][0], test_points[1][1], "k*", markersize=16)
    plt.plot(test_points[2][0], test_points[2][1], "k*", markersize=16)
    plt.plot(test_points[3][0], test_points[3][1], "k*", markersize=16)
```

- How would you classify the following points?
  - Pick the class with the highest value for the classification formula.

In [8]:
```
# You don't have to understand the code.
discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(lr.coef_, lr.intercept_, cm3.colors):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plot_test_points()
plt.ylim(-10, 15)
plt.xlim(-10, 8)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(
    ["Class 0", "Class 1", "Class 2", "Line class 0", "Line class 1", "
    loc=(1.01, 0.3),
);
```



In [9]:
```
# You don't have to understand the code below.
plot_2d_classification(lr, X_train, fill=True, alpha=0.7)
discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(lr.coef_, lr.intercept_, cm3.colors):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plot_test_points()
plt.legend(
    ["Class 0", "Class 1", "Class 2", "Line class 0", "Line class 1", "
    loc=(1.01, 0.3),
)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1");
```




In [10]:
```
test_points = [[-4.0, 12], [-2, 0.0], [-8, 3.0], [4, 8.5]]
lr.predict_proba(test_points)
```

Out[10]:
```
array([[9.99645770e-01, 1.77111249e-04, 1.77118671e-04],
       [4.92621189e-02, 2.36353992e-01, 7.14383889e-01],
       [6.11446337e-01, 6.67363501e-06, 3.88546990e-01],
       [4.26997745e-01, 5.72993816e-01, 8.43813466e-06]])
```

In [11]:
```
lr.predict(test_points)
```

Out[11]:
```
array([0, 2, 0, 1])
```

## One Vs. One approach

- Build a binary model for each pair of classes.
- 1v2, 1v3, 2v3
- Trains $\frac{n \times (n-1)}{2}$ binary classifiers

- Trained on relatively balanced subsets

## One Vs. One prediction

- Apply all of the classifiers on the test example.
- Count how often each class was predicted.
- Predict the class with most votes.

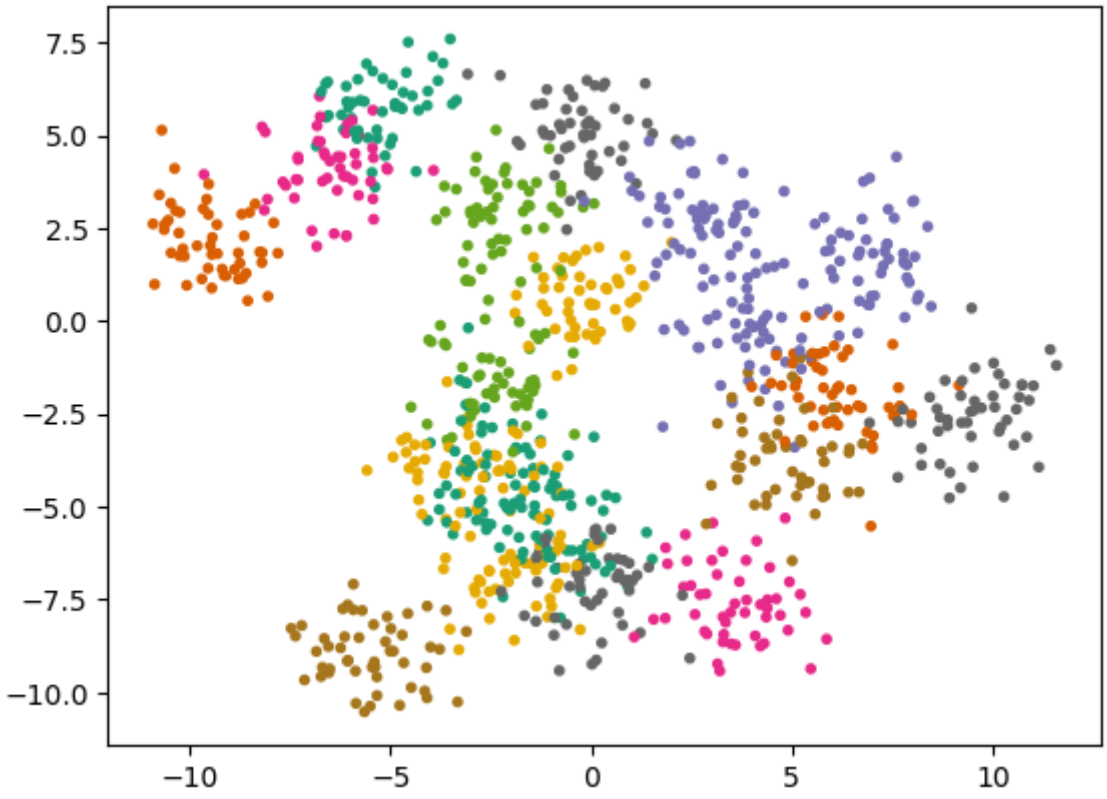
## Using OVR and OVO as wrappers

- You can use these strategies as meta-strategies for any binary classifiers.
  - `OneVsRestClassifier` (https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html)
  - `OneVsOneClassifier` (https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsOneClassifier.html)

- When do we use `OneVsRestClassifier` and `OneVsOneClassifier`
- It's not that likely for you to need `OneVsRestClassifier` or `OneVsOneClassifier` because most of the methods you'll use will have native multi-class support.
- However, it's good to know in case you ever need to extend a binary classifier (perhaps one you've implemented on your own).

In [12]:
```
from sklearn.multiclass import OneVsOneClassifier, OneVsRestClassifier
```

In [13]:
```
# Let's examine the time taken by OneVsRestClassifier and OneVsOneClass

# generate blobs with fixed random generator
X_multi, y_multi = make_blobs(n_samples=1000, centers=20, random_state=

X_train_multi, X_test_multi, y_train_multi, y_test_multi = train_test_s
    X_multi, y_multi
)

plt.scatter(*X_multi.T, c=y_multi, marker=".", cmap="Dark2");
```




In [14]:
```
model = OneVsOneClassifier(LogisticRegression())
%timeit model.fit(X_train_multi, y_train_multi);
print("With OVO wrapper")
print(model.score(X_train_multi, y_train_multi))
print(model.score(X_test_multi, y_test_multi))
```

```
209 ms ± 8.73 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
With OVO wrapper
0.8066666666666666
0.756
```

In [15]:
```
model = OneVsRestClassifier(LogisticRegression())
%timeit model.fit(X_train_multi, y_train_multi);
print("With OVR wrapper")
print(model.score(X_train_multi, y_train_multi))
print(model.score(X_test_multi, y_test_multi))
```

```
29.2 ms ± 1.49 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
With OVR wrapper
0.7546666666666667
0.668
```

- As expected OVO takes more time compared to OVR.
- Here (https://scikit-learn.org/stable/modules/multiclass.html) you will find summary of how `scikit-learn` handles multi-class classification for different classifiers.

## True/False

1. One-vs.-one strategy uses all the available data when training each binary classifier.
2. For a 100-class classification problem, one-vs.-rest multi-class strategy will create 100 binary classifiers.

## Multi-class classification on HappyDB (https://www.kaggle.com/ritresearch/happydb) corpus

Let's examine precision, recall, and f1-score of different classes in the HappyDB (https://www.kaggle.com/ritresearch/happydb) corpus.

In [16]:
```
df = pd.read_csv("../data/cleaned_hm.csv", index_col=0)
sample_df = df.dropna()
sample_df.head()
sample_df = sample_df.rename(
    columns={"cleaned_hm": "moment", "ground_truth_category": "target"}
)
sample_df.head()
```

Out[16]:

| hmid | wid | reflection_period | original_hm | moment | modified | num_sentence | target | predicte |
|---|---|---|---|---|---|---|---|---|
| **27676** | 206 | 24h | We had a serious talk with some friends of our... | We had a serious talk with some friends of our... | True | 2 | bonding | |
| **27678** | 45 | 24h | I meditated last night. | I meditated last night. | True | 1 | leisure | |
| **27697** | 498 | 24h | My grandmother start to walk from the bed afte... | My grandmother start to walk from the bed afte... | True | 1 | affection | |
| **27705** | 5732 | 24h | I picked my daughter up from the airport and w... | I picked my daughter up from the airport and w... | True | 1 | bonding | |
| **27715** | 2272 | 24h | when i received flowers from my best friend | when i received flowers from my best friend | True | 1 | bonding | |

In [17]:
```
sample_df["target"].value_counts()
```

Out[17]:
```
affection           4810
achievement         4276
bonding             1750
enjoy_the_moment    1514
leisure             1306
nature               252
exercise             217
Name: target, dtype: int64
```

It's a multiclass classification problem!

In [18]:
```
train_df, test_df = train_test_split(sample_df, test_size=0.3, random_s
X_train_happy, y_train_happy = train_df["moment"], train_df["target"]
X_test_happy, y_test_happy = test_df["moment"], test_df["target"]
```

Typesetting math: 100%

In [19]:
```
from sklearn.feature_extraction.text import CountVectorizer

pipe_lr = make_pipeline(
    CountVectorizer(stop_words="english"), LogisticRegression(max_iter=
)
```

In [20]:
```
pipe_lr.fit(X_train_happy, y_train_happy)
```

Out[20]:
```
Pipeline(steps=[('countvectorizer', CountVectorizer(stop_words='englis
h')),
                ('logisticregression', LogisticRegression(max_iter=200
0))])
```
**In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.**
**On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.**

In [21]:
```
preds = pipe_lr.predict(X_test_happy)[:5]
preds
```

Out[21]:
```
array(['achievement', 'affection', 'bonding', 'enjoy_the_moment',
       'affection'], dtype=object)
```

Note that the output of `predict_proba` now contains a probability for each class:

In [22]:
```
pipe_lr.predict_proba(X_test_happy)[:5]
```

Out[22]:
```
array([[7.06472042e-01, 3.74986136e-02, 5.65462935e-02, 4.48416453e-02,
        3.05695812e-02, 1.10293654e-01, 1.37781706e-02],
       [4.51114062e-03, 9.89340017e-01, 5.83647210e-04, 3.70764136e-03,
        2.90911962e-04, 6.37035303e-04, 9.29606301e-04],
       [2.13299803e-03, 1.51563896e-02, 9.78842547e-01, 1.36505813e-03,
        1.26043214e-03, 9.41950993e-04, 3.00623854e-04],
       [1.13093070e-01, 9.17955178e-02, 2.38830890e-02, 5.06663413e-01,
        7.31006757e-03, 2.49573509e-01, 7.68133360e-03],
       [7.72068573e-02, 5.53170559e-01, 3.87164821e-02, 8.14773263e-02,
        2.38206330e-02, 2.08447747e-01, 1.71603952e-02]])
```

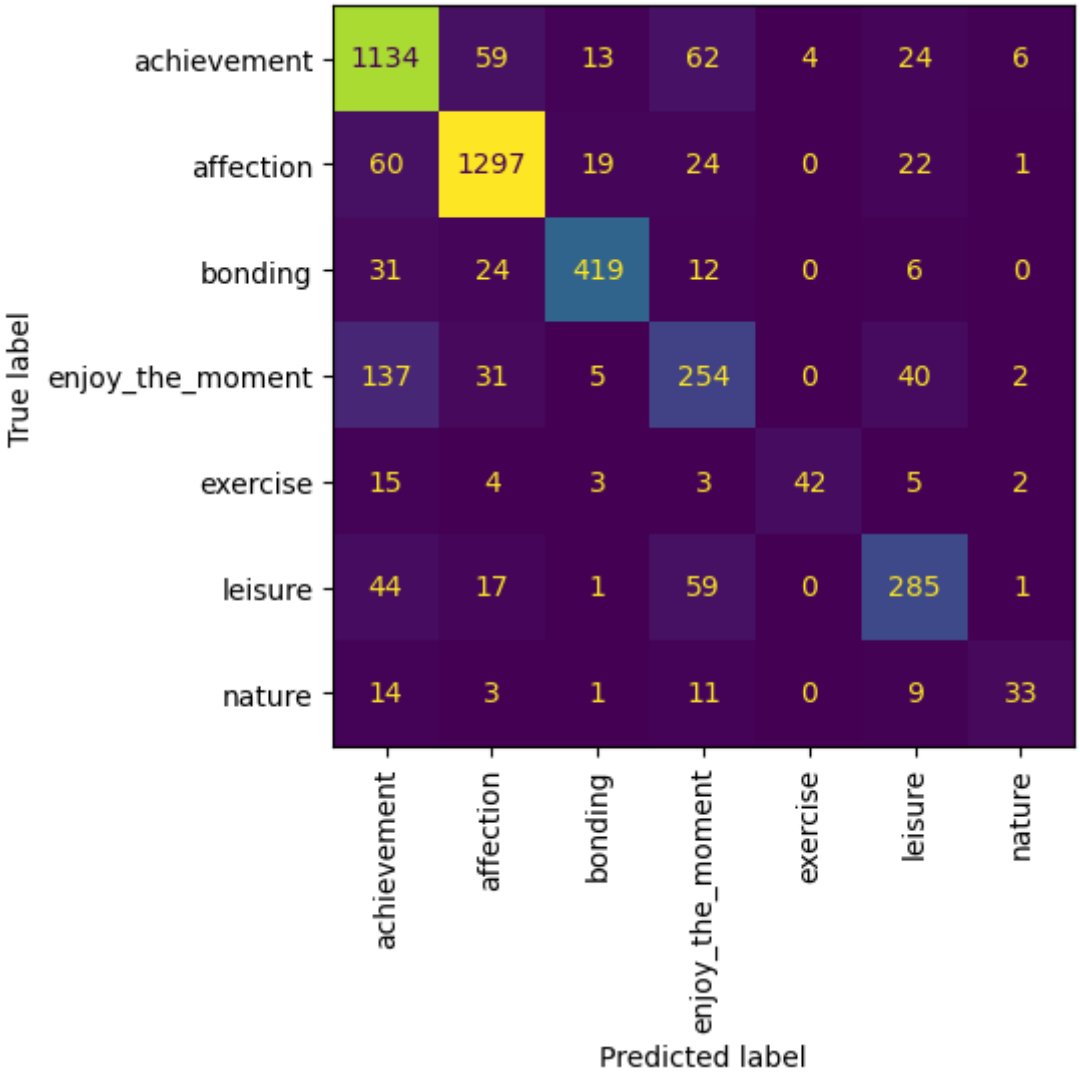
And you'll see that each row adds up to 1, as expected:

In [23]:
```
pipe_lr.predict_proba(X_test_happy).sum(axis=1)
```

Out[23]:
```
array([1., 1., 1., ..., 1., 1., 1.])
```

We can also make a confusion matrix:

In [24]:
```
from sklearn.metrics import plot_confusion_matrix, ConfusionMatrixDispl

ConfusionMatrixDisplay.from_estimator(pipe_lr,
    X_test_happy,
    y_test_happy,
    values_format="d",
    xticks_rotation="vertical",
    colorbar=False,
);
```




And print the classification report.

In [25]:
```
print(classification_report(y_test_happy, pipe_lr.predict(X_test_happy)
```

```
                  precision    recall  f1-score   support

     achievement       0.79      0.87      0.83      1302
       affection       0.90      0.91      0.91      1423
         bonding       0.91      0.85      0.88       492
enjoy_the_moment       0.60      0.54      0.57       469
        exercise       0.91      0.57      0.70        74
         leisure       0.73      0.70      0.71       407
          nature       0.73      0.46      0.57        71

        accuracy                           0.82      4238
       macro avg       0.80      0.70      0.74      4238
    weighted avg       0.82      0.82      0.81      4238
```

- Seems like there is a lot of variation in the scores for different classes.
- The model is performing pretty well on *affection* class but not that well on *enjoy_the_moment* and *nature* classes.

How are the predictions made?

In [26]:
```
pipe_lr.predict_proba(X_test_happy)[:5]
```

Out[26]:
```
array([[7.06472042e-01, 3.74986136e-02, 5.65462935e-02, 4.48416453e-02,
        3.05695812e-02, 1.10293654e-01, 1.37781706e-02],
       [4.51114062e-03, 9.89340017e-01, 5.83647210e-04, 3.70764136e-03,
        2.90911962e-04, 6.37035303e-04, 9.29606301e-04],
       [2.13299803e-03, 1.51563896e-02, 9.78842547e-01, 1.36505813e-03,
        1.26043214e-03, 9.41950993e-04, 3.00623854e-04],
       [1.13093070e-01, 9.17955178e-02, 2.38830890e-02, 5.06663413e-01,
        7.31006757e-03, 2.49573509e-01, 7.68133360e-03],
       [7.72068573e-02, 5.53170559e-01, 3.87164821e-02, 8.14773263e-02,
        2.38206330e-02, 2.08447747e-01, 1.71603952e-02]])
```

In [27]:
```
np.argmax(pipe_lr.predict_proba(X_test_happy), axis=1)
```

Out[27]:
```
array([0, 1, 2, ..., 1, 0, 2])
```

In [28]:
```
classes = pipe_lr.classes_
classes
```

Out[28]:
```
array(['achievement', 'affection', 'bonding', 'enjoy_the_moment',
       'exercise', 'leisure', 'nature'], dtype=object)
```

In [29]:
```
y_hat = pipe_lr.predict(X_test_happy)
y_hat
```

Out[29]:
```
array(['achievement', 'affection', 'bonding', ..., 'affection',
       'achievement', 'bonding'], dtype=object)
```

How many coefficients have we learned?

In [30]:
```
pipe_lr.named_steps["logisticregression"].coef_.shape
```

Out[30]:
```
(7, 8060)
```

- We have one coefficient per feature *per class*.
- Let's examine them.

In [31]:
```
feature_names = pipe_lr.named_steps["countvectorizer"].get_feature_name
lr_coefs = pd.DataFrame(
    data=pipe_lr.named_steps["logisticregression"].coef_.T,
    index=feature_names,
    columns=classes,
).sort_values("bonding", ascending=False)
lr_coefs
```

Out[31]:

| | achievement | affection | bonding | enjoy_the_moment | exercise | leisure | nature |
|---|---|---|---|---|---|---|---|
| **friend** | -1.687528 | -0.183608 | 5.589866 | -1.707829 | 0.330456 | -1.769253 | -0.572105 |
| **friends** | -1.304287 | 0.052779 | 5.246125 | -1.992716 | 0.328815 | -1.559656 | -0.771061 |
| **roommate** | -1.327202 | -0.690686 | 3.418085 | -1.138203 | -0.078646 | -0.070284 | -0.113063 |
| **coworkers** | -0.588496 | -0.606175 | 3.011513 | -1.098932 | -0.088536 | -0.518752 | -0.110623 |
| **coworker** | -0.934857 | -0.591312 | 2.770930 | -0.560366 | -0.093246 | -0.415496 | -0.175652 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **feelings** | 0.057296 | 1.194994 | -0.898329 | -0.123828 | -0.103999 | -0.093033 | -0.033103 |
| **jogging** | -0.046343 | -0.319887 | -0.909238 | -0.098763 | 1.521341 | -0.130390 | -0.016719 |
| **telling** | -0.436909 | 0.870929 | -1.070986 | 0.694175 | -0.066820 | 0.038809 | -0.029197 |
| **drive** | -0.150838 | 0.584506 | -1.184986 | 0.891987 | -0.239585 | -0.453318 | 0.552235 |
| **boy** | 1.398748 | 0.288127 | -1.327434 | 0.138895 | -0.261471 | -0.162182 | -0.074683 |

8060 rows × 7 columns

The interpretation is a feature importance for predicting a certain class. For example:

In [32]:
```
lr_coefs.loc["friend"][2]
```

Out[32]:
```
5.589866397597406
```

- This means that if the value for the feature "friend" is bigger, you are more likely to predict class "bonding".

- If you want a general feature importance irrespective of class, you could try looking at the sum of the squares of the coefficients, which is what sklearn does:

In [33]:
```
(lr_coefs ** 2).sum(axis=1).sort_values(ascending=False)
```

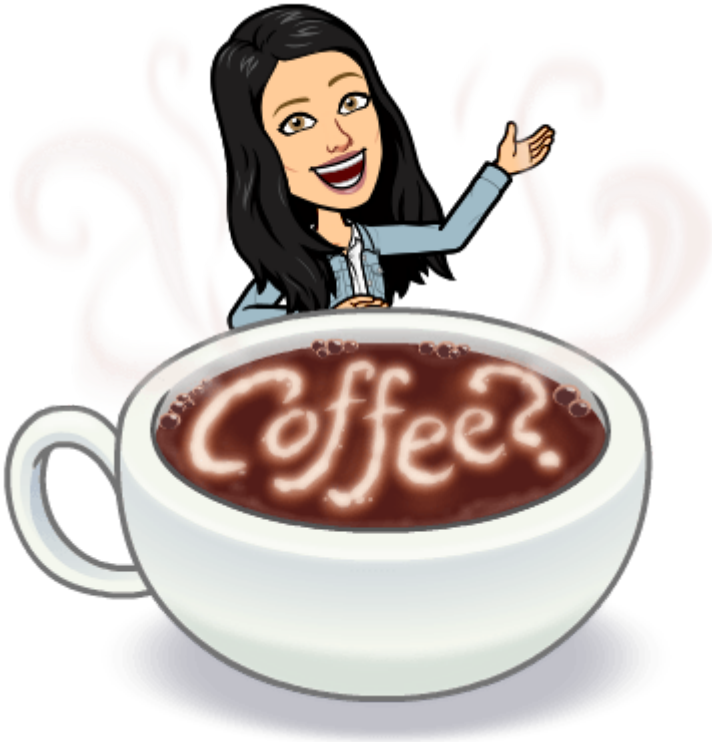
Out[33]:
```
friend        4.061151e+01
friends       3.633188e+01
husband       2.764409e+01
wife          2.542235e+01
son           2.361098e+01
                  ...
curiosity     1.996211e-11
gang          1.728753e-11
teases        1.728753e-11
horror        1.728753e-11
cinemas       1.728753e-11
Length: 8060, dtype: float64
```

In [34]:
```
?LogisticRegression
```

```
Init signature:
LogisticRegression(
    penalty='l2',
    *,
    dual=False,
    tol=0.0001,
    C=1.0,
    fit_intercept=True,
    intercept_scaling=1,
    class_weight=None,
    random_state=None,
    solver='lbfgs',
    max_iter=100,
    multi_class='auto',
    verbose=0,
    warm_start=False,
    n_jobs=None,
    l1_ratio=None,
)
Docstring:
```

- We can see that there's a `multi_class` parameter, that can be set to `'ovr'` or `'multinomial'`, or you can have it automatically choose between the two, which is the default.
  - In CPSC 340 we discuss in detail the difference between these two approaches.
  - In CPSC 340 we make an argument for preferring `'multinomial'`, but in short it doesn't matter which one you choose.

## Break (5 min)

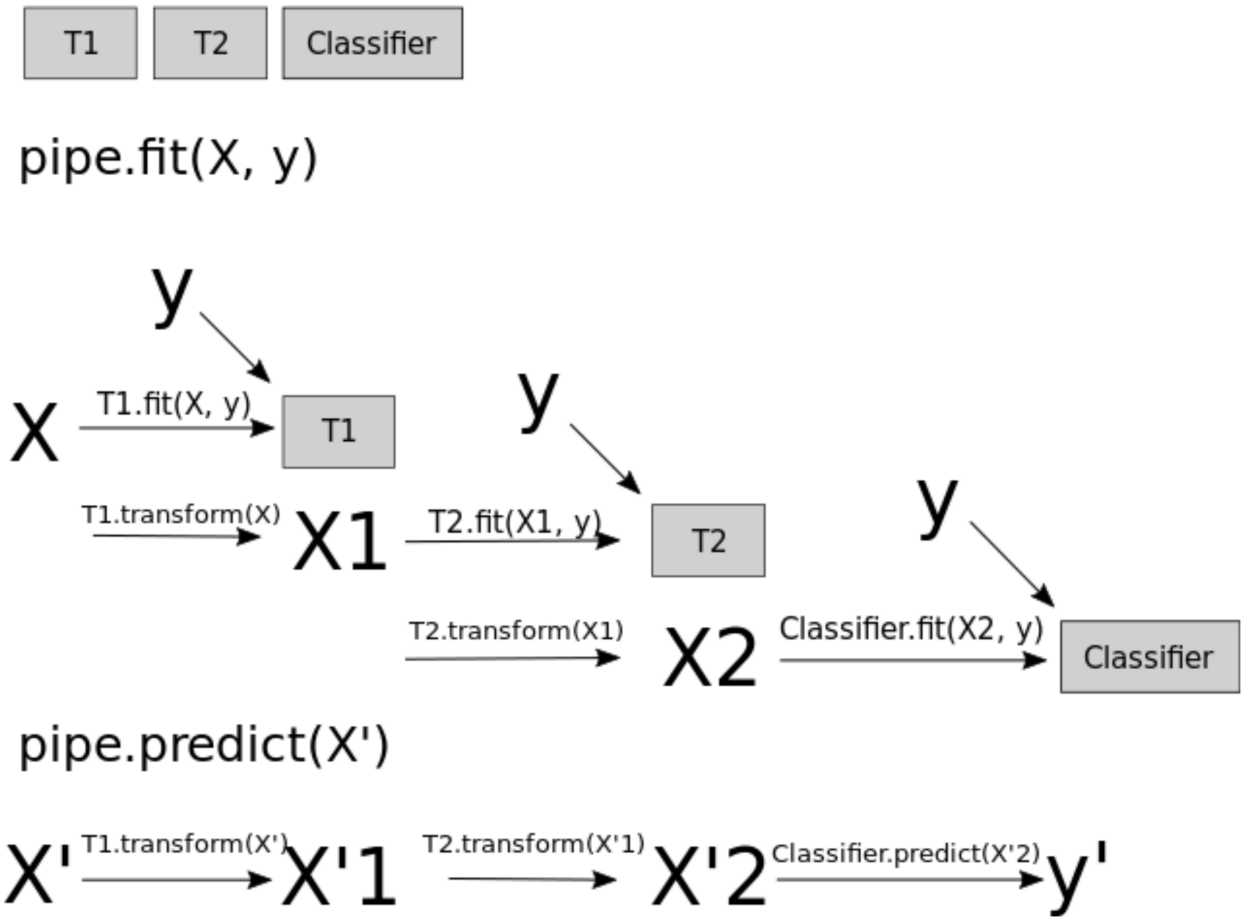



## Intro to computer vision

- Computer vision (https://en.wikipedia.org/wiki/Computer_vision) refers to understanding images/videos, usually using ML/AI.
- Computer vision has many tasks of interest:
  - image classification: is this a cat or a dog?
  - object localization: where are the people in this image?
  - image segmentation: what are the various parts of this image?
  - motion detection: what moved between frames of a video?
  - and much more...
- We will focus on image classification.

## Intro to neural networks

- Very popular these days under the name **deep learning**.
- Neural networks apply a sequence of transformations on your input data.
- At a very high level you can think of them as `Pipelines` in `sklearn`.
- A neural network is a model that's sort of like its own pipeline
  - It involves a series of transformations ("layers") internally.
  - The output is the prediction.




Source (https://amueller.github.io/COMS4995-s20/slides/aml-04-preprocessing/#18)

- They can be viewed a generalization of linear models where we apply a series of transformations.
- Here is graphical representation of logistic regression model.
  - We have 4 features: x[0], x[1], x[2], x[3]

In [35]:
```
from mglearn_utils import plot_logistic_regression_graph, plot_single_h

display(plot_logistic_regression_graph())
```

```
<graphviz.graphs.Digraph at 0x196ce9b70>
```

- Below we are adding one "layer" of transformations in between features and the target.
- We are repeating the the process of computing the weighted sum multiple times.
- The **hidden units** (e.g., h[1], h[2], ...) represent the intermediate processing steps.

In [36]:
```
display(plot_single_hidden_layer_graph())
```

```
<graphviz.graphs.Digraph at 0x196cea650>
```

- Now we are adding one more layer of transformations.

In [37]:
```
1 display(plot_two_hidden_layer_graph())
```

```
<graphviz.graphs.Digraph at 0x196dc1540>
```

- Important question: how many features before/after transformation.
  - e.g. scaling doesn't change the number of features
  - OHE increases the number of features
- With a neural net, you specify the number of features after each transformation.
  - In the above, it goes from 4 to 3 to 3 to 1 (output).

- To make them really powerful compared to the linear models, we apply a non-linear function to the weighted sum for each hidden node.

## Terminology

- Neural network = neural net
- Deep learning ~ using neural networks

## Why neural networks?

- They can learn very complex functions.
  - The fundamental tradeoff is primarily controlled by the **number of layers** and **layer sizes**.
  - More layers / bigger layers --> more complex model.
  - You can generally get a model that will not underfit.

## Why neural networks?

- The work really well for structured data:
  - 1D sequence, e.g. timeseries, language
  - 2D image
  - 3D image or video
- They've had some incredible successes in the last 10 years.
- Transfer learning (coming later today) is really useful.

## Why not neural networks?

- Often they require a lot of data.
- They require a lot of compute time, and, to be faster, specialized hardware called GPUs (https://en.wikipedia.org/wiki/Graphics_processing_unit).
- They have huge numbers of hyperparameters and are a huge pain to tune.
  - Think of each layer having hyperparameters, plus some overall hyperparameters.
  - Being slow compounds this problem.
- They are not interpretable.

### Why not neural networks?

- When you call `fit`, you are not guaranteed to get the optimal.
  - There are now a bunch of hyperparameters specific to `fit`, rather than the model.
  - You never really know if `fit` was successful or not.
  - You never really know if you should have run `fit` for longer.
- I don't recommend training them on your own without further training
  - Take CPSC 340 and other courses if you're interested.
  - I'll show you some ways to use neural networks without calling `fit`.

### Deep learning software

- scikit-learn has MLPRegressor (https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html) and MLPClassifier (https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html) but they aren't very flexible.
  - In general you'll want to leave the scikit-learn ecosystem when using neural networks.
  - Fun fact: these classes were contributed to scikit-learn by a UBC graduate student.
- There's been a lot of deep learning software out there.

- The current big players are:

1. TensorFlow (https://www.tensorflow.org)
2. PyTorch (http://pytorch.org)

- Both are heavily used in industry.
- If interested, see comparison of deep learning software (https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software).

# Neural networks on image data

In [38]:
```
import matplotlib as mpl
from sklearn.datasets import fetch_lfw_people

mpl.rcParams.update(mpl.rcParamsDefault)
plt.rcParams["image.cmap"] = "gray"
```

In [39]:
```
from sklearn.datasets import fetch_lfw_people

people = fetch_lfw_people(min_faces_per_person=40, resize=0.7)

fig, axes = plt.subplots(2, 5, figsize=(12, 6), subplot_kw={"xticks": (
for target, image, ax in zip(people.target, people.images, axes.ravel()
    ax.imshow(image)
    ax.set_title(people.target_names[target])

plt.show();
```


In [40]:
```
image_shape = people.images[0].shape
print("people.images.shape: {}".format(people.images.shape))
print("Number of classes: {}".format(len(people.target_names)))
```

```
people.images.shape: (1867, 87, 65)
Number of classes: 19
```

There are 1,867 images stored as arrays of 5655 pixels (87 by 65), of 19 different people:

In [41]:
```
# count how often each target appears
counts = np.bincount(people.target)
df = pd.DataFrame(counts, columns=["count"], index=people.target_names)
df.sort_values("count", ascending=False)
```

Out[41]:

| | count |
|---|---|
| George W Bush | 530 |
| Colin Powell | 236 |
| Tony Blair | 144 |
| Donald Rumsfeld | 121 |
| Gerhard Schroeder | 109 |
| Ariel Sharon | 77 |
| Hugo Chavez | 71 |
| Junichiro Koizumi | 60 |
| Jean Chretien | 55 |
| John Ashcroft | 53 |
| Jacques Chirac | 52 |
| Serena Williams | 52 |
| Vladimir Putin | 49 |
| Luiz Inacio Lula da Silva | 48 |
| Gloria Macapagal Arroyo | 44 |
| Arnold Schwarzenegger | 42 |
| Jennifer Capriati | 42 |
| Laura Bush | 41 |
| Lleyton Hewitt | 41 |

Let's make the data less skewed by taking only 20 images of the each person.

In [42]:
```
mask = np.zeros(people.target.shape, dtype=bool)
for target in np.unique(people.target):
    mask[np.where(people.target == target)[0][:20]] = 1

X_people = people.data[mask]
y_people = people.target[mask]
```

In [43]:
```
X_people.shape
```

Out[43]: (380, 5655)

Typesetting math: 100%

In [44]:
```
# scale the grayscale values to be between 0 and 1
# instead of 0 and 255 for better numeric stability
X_people = X_people / 255.0
```

In [45]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X_people, y_people, random_state=123
)
```

In [46]:
```
X_train
```

Out[46]:
```
array([[0.0000000e+00, 0.0000000e+00, 0.0000000e+00, ..., 0.0000000e+00,
        0.0000000e+00, 0.0000000e+00],
       [0.0000000e+00, 0.0000000e+00, 0.0000000e+00, ..., 0.0000000e+00,
        0.0000000e+00, 0.0000000e+00],
       [5.1262336e-06, 0.0000000e+00, 0.0000000e+00, ..., 0.0000000e+00,
        0.0000000e+00, 0.0000000e+00],
       ...,
       [0.0000000e+00, 5.1262336e-06, 1.0252467e-05, ..., 0.0000000e+00,
        0.0000000e+00, 0.0000000e+00],
       [0.0000000e+00, 0.0000000e+00, 0.0000000e+00, ..., 9.1246958e-04,
        7.2279893e-04, 7.1254646e-04],
       [1.2354223e-03, 1.2354223e-03, 1.2405486e-03, ..., 1.5840061e-03,
        1.5532487e-03, 1.5532487e-03]], dtype=float32)
```

Now the data is in this tabular format that we are used to. Now we can use our usual classification methods.

In [47]:
```
lr = LogisticRegression(max_iter=4000)
```

In [48]:
```
lr.fit(X_train, y_train);
```

In [49]:
```
lr.score(X_train, y_train)
```

Out[49]:
```
0.07017543859649122
```

In [50]:
```
lr.score(X_test, y_test)
```

Out[50]:
```
0.010526315789473684
```

We are getting very poor test results :(.

- Why flattening images is a bad idea?
  - By "flattening" the image we throw away useful information.
- What the computer sees:

In [51]:
```
list(X_train[0])[150:200]
```

Out[51]:
```
[1.0252467e-05,
 1.0252467e-05,
 5.1262336e-06,
 1.0252467e-05,
 5.1262336e-06,
 5.1262336e-06,
 5.1262336e-06,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 5.638857e-05,
 5.638857e-05,
 0.0005895169,
 0.0007227989,
 0.00095860567,
 0.0017788031,
 0.0017531719,
 0.0022657954,
 0.0027015249,
 0.0027681661,
 0.003142381,
 0.0031526333,
 0.0031372549,
 0.0031577598,
 0.0031526333,
 0.0030859925,
 0.0030654876,
 0.0030706136,
 0.0030193515,
 0.0030039728,
 0.0029937204,
 0.0030603614,
 0.0021376396,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0]
```

- Hard to classify this!
- Convolutional neural networks (https://en.wikipedia.org/wiki/Convolutional_neural_network) (CNNs) can take in images without flattening them.

- We won't cover CNNs here, but they are in CPSC 340.

# Transfer learning

- In practice, very few people train an entire CNN from scratch because it requires a large dataset, powerful computers, and a huge amount of human effort to train the model.
- Instead, a common practice is to download a pre-trained model and fine tune it for your task.
- This is called **transfer learning**.
- Transfer learning is one of the most common techniques used in the context of computer vision and natural language processing.
  - In the last lecture we used pre-trained embeddings to train create text representation.

## Using pre-trained models out-of-the-box

Recall this example I showed you in the intro video (our very first lecture).

In [52]:
```
def classify_image(img, topn=4):
    clf = vgg16(weights=VGG16_Weights.DEFAULT)  # Loading the pre-train
    preprocess = transforms.Compose(
        [
            transforms.Resize(299),
            transforms.CenterCrop(299),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229
        ]
    )  # Defining a preprocessor to transform a given image so that it'

    with open("../data/imagenet_classes.txt") as f:
        classes = [line.strip() for line in f.readlines()]

    img_t = preprocess(img)
    batch_t = torch.unsqueeze(img_t, 0)
    clf.eval()
    output = clf(batch_t)
    _, indices = torch.sort(output, descending=True)
    probabilities = torch.nn.functional.softmax(output, dim=1)
    d = {
        "Class": [classes[idx] for idx in indices[0][:topn]],
        "Probability score": [
            np.round(probabilities[0, idx].item(), 3) for idx in indice
        ],
    }
    df = pd.DataFrame(d, columns=["Class", "Probability score"])
    return df
```

In [53]:
```
import torch
from PIL import Image
from torchvision import transforms
from torchvision.models import vgg16, VGG16_Weights, DenseNet121_Weight
```

In [54]:
```
# Predict labels with associated probabilities for unseen images
images = glob.glob("../data/test_images/*.*")
for image in images:
    img = Image.open(image)
    img.load()
    plt.imshow(img)
    plt.show()
    df = classify_image(img)
    print(df.to_string(index=False))
    print("--------------------------------------------------------------------
```


- We got these predictions without "doing the ML ourselves".
- We are using **pre-trained** `vgg16` model which is available in `torchvision`
- `torchvision` has many such pre-trained models available that have been very successful across a wide range of tasks: AlexNet, VGG, ResNet, Inception, MobileNet, etc.
- Many of these models have been pre-trained on famous datasets like **ImageNet**.

## ImageNet

- ImageNet (http://www.image-net.org/) is an image dataset that became a very popular benchmark in the field ~10 years ago.
- Wikipedia article (https://en.wikipedia.org/wiki/ImageNet)
- There are 14 million images and 1000 classes.
- Here are some example classes.

In [55]:
```
with open("../data/imagenet_classes.txt") as f:
    classes = [line.strip() for line in f.readlines()]
classes[100:110]
```

Out[55]:
```
['black swan, Cygnus atratus',
 'tusker',
 'echidna, spiny anteater, anteater',
 'platypus, duckbill, duckbilled platypus, duck-billed platypus, Ornithor
hynchus anatinus',
 'wallaby, brush kangaroo',
 'koala, koala bear, kangaroo bear, native bear, Phascolarctos cinereus',
 'wombat',
 'jellyfish',
 'sea anemone, anemone',
 'brain coral']
```

Let's see what labels this pre-trained model give us for CPSC 330 teaching team.

In [56]:
```
# Predict labels with associated probabilities for unseen images
images = glob.glob("../data/_330_teaching_team/*.*")
for image in images:
    img = Image.open(image)
    img.load()
    plt.imshow(img)
    plt.show()
    df = classify_image(img)
    print(df.to_string(index=False))
    print("-------------------------------------------------------------------
```


- It's not doing very well here because ImageNet don't have classes for Giulia, Michelle, Amir, or Mathias.
- Here we are using pre-trained models out-of-the-box.
- Can we use pre-trained models for our own classification problem with our classes?

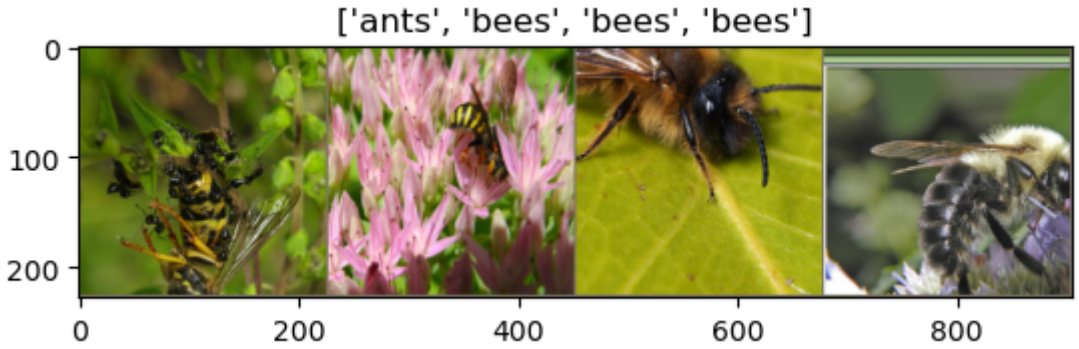
## Using pre-trained models as feature extractor

- Here we will use pre-trained models to extract features.
- We will pass our specific data through a pre-trained network to get a feature vector for each example in the data.
- You train a machine learning classifier such as logistic regression or random forest using these extracted feature vectors.

In [57]:

```
# Attribution: [Code from PyTorch docs](https://pytorch.org/tutorials/b

import copy
import os
import time

import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torch.optim import lr_scheduler
from torchvision import datasets, models, transforms

data_transforms = {
    "train": transforms.Compose(
        [
            transforms.RandomResizedCrop(224),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
        ]
    ),
    "val": transforms.Compose(
        [
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
        ]
    ),
}
data_dir = "../data/hymenoptera_data"
image_datasets = {
    x: datasets.ImageFolder(os.path.join(data_dir, x), data_transforms[
    for x in ["train", "val"]
}
dataloaders = {
    x: torch.utils.data.DataLoader(
        image_datasets[x], batch_size=4, shuffle=True, num_workers=4
    )
    for x in ["train", "val"]
}
dataset_sizes = {x: len(image_datasets[x]) for x in ["train", "val"]}
class_names = image_datasets["train"].classes

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

In [58]:
```
def imshow(inp, title=None):
    """Imshow for Tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.pause(0.001)  # pause a bit so that plots are updated


# Get a batch of training data
inputs, classes = next(iter(dataloaders["train"]))

# Make a grid from batch
out = torchvision.utils.make_grid(inputs)

imshow(out, title=[class_names[x] for x in classes])
```




In [59]:
```
print(f"Classes: {image_datasets['train'].classes}")
print(
    f"Class count: {image_datasets['train'].targets.count(0)}, {image_d
)
print(f"Samples:", len(image_datasets["train"]))
print(f"First sample: {image_datasets['train'].samples[0]}")
```

```
Classes: ['ants', 'bees']
Class count: 123, 121
Samples: 244
First sample: ('../data/hymenoptera_data/train/ants/0013035.jpg', 0)
```

In [60]:
```
def get_features(model, train_loader, valid_loader):
    """Extract output of squeezenet model"""

    with torch.no_grad():  # turn off computational graph stuff
        Z_train = torch.empty((0, 1024))  # Initialize empty tensors
        y_train = torch.empty((0))
        Z_valid = torch.empty((0, 1024))
        y_valid = torch.empty((0))
        for X, y in train_loader:
            Z_train = torch.cat((Z_train, model(X)), dim=0)
            y_train = torch.cat((y_train, y))
        for X, y in valid_loader:
            Z_valid = torch.cat((Z_valid, model(X)), dim=0)
            y_valid = torch.cat((y_valid, y))
    return Z_train.detach(), y_train.detach(), Z_valid.detach(), y_vali
```

In [61]:
```
densenet = models.densenet121(weights=DenseNet121_Weights.DEFAULT)
densenet.classifier = nn.Identity()  # remove that last "classification
```

In [ ]:
```
Z_train, y_train, Z_valid, y_valid = get_features(
    densenet, dataloaders["train"], dataloaders["val"]
)
```

Now we have some extracted features.

In [ ]:
```
Z_train.shape
```

In [ ]:
```
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import StandardScaler

pipe = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000
pipe.fit(Z_train, y_train)
pipe.score(Z_train, y_train)
```

In [ ]:
```
pipe.score(Z_valid, y_valid)
```

- This is great accuracy for so little data (We only have 244 examples.) and little effort!!!

## TODO

- Compare this to accuracy with flattened images and logistic regression
- Try this out with the Faces dataset.

# Random cool stuff

- Style transfer: given a "content image" and a "style image", create a new image with the content of one and the style of the other.
  - Here is the original paper from 2015 (https://arxiv.org/pdf/1508.06576.pdf), see Figure 2.

    - Here are more in this 2016 paper (https://arxiv.org/pdf/1601.04589.pdf); see, e.g. Figures 1 and 7.
    - This has been done for video as well; see this video from 2016 (https://www.youtube.com/watch?v=Khuj4ASldmU).
- Image captioning (https://cs.stanford.edu/people/karpathy/sfmltalk.pdf): Transfer learning with NLP and vision
- Colourization: see this 2016 project (http://iizuka.cs.tsukuba.ac.jp/projects/colorization/en/).
- Inceptionism: let the neural network "make things up"
    - 2015 article (https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html)
    - "Deep dream" video from 2015 (https://www.youtube.com/watch?v=dbQh1I_uvjo).

## Summary

- Multi-class classification refers to classification with >2 classes.
    - Most sklearn classifiers work out of the box.
    - With `LogisticRegression` the situation with the coefficients is a bit funky, we get 1 coefficient per feature per class.
- Flattening images throws away a lot of useful information (sort of like one-hot encoding on ordinal variable!).
- Neural networks are a flexible class of models.
    - They are hard to train - a lot more on that in CPSC 340.
    - They generally require leaving the sklearn ecosystem to tensorflow or pytorch.
    - They are particular powerful for structured input like images, videos, audio, etc.
- The good news is we can use pre-trained neural networks.
    - This saves us a huge amount of time/cost/effort/resources.
    - We can use these pre-trained networks directly or use them as feature transformers.
- My general recommendation: don't use deep learning unless there is good reason to.